

The Dreaded Command Line

May 19, 2003 at 23:55, by Geoff Richards¹

This is an expanded prose version of a short talk on the Unix command line² I gave a while back on behalf of GBdirect³. A nice PDF version of the slides⁴ is available.

The Unix and Linux command line

Unix has always been considered a ‘command line’ operating system—and although modern GNU/Linux distributions come with slick graphical interfaces, there's still a place for the ‘old fashioned’ way of doing things.

In the early days of Unix (the early 1970s) there was no choice but to use the command line. The minicomputers that Unix ran on were big and slow, and only talked to their users through clunky **dumb terminals**, which of course didn't support graphics. But programmers and system administrators still use the command line, often only using a **graphical user interface** (GUI) for web browsing and image editing. This article will look at some of the reasons why this is so, and get you started on using the command line.

Pros and Cons

The command line seems slow and inefficient to people who are starting out with it, but with a bit of practice it can be much quicker to use than GUI interfaces, at least for many common tasks. The keyboard can be an efficient way of communicating with a computer, whereas the mouse is much more limited in the amount of information it can convey—a few clicks per second. Even simple tasks like running a GUI program can be quicker—typing ‘mozilla’ to run a web browser is usually quicker than finding it deep in a menu.

Typed commands can also express a lot of complex information fairly concisely. The real power of the command line interface is in tying programs together, or running a whole set of related commands in one go. And nor is it as tedious to type commands as once it used to be. Modern Unix command line interfaces have features designed to help you type commands and correct mistakes.

This is not to say that the GUI doesn't have a place. Some things, like editing pictures, are certainly better suited to a graphical interface, and non-technical users who just want to do some word-processing or check their email will be better off with them. It takes time to learn the command line well enough to get the benefit of it, so it's not for everyone.

How it all fits together

Your immediate interface to the command line is through a **terminal**. This is what reads the things you type and displays the text output of the programs you run.

When you've typed a command and pressed `Enter` to run it, a program called the **shell** takes it and decides what to do about it. Typically this involves running one or more other programs—and Unix or Linux machines have a vast array of command line programs you might want to run.

The whole setup looks something like this:

The terminal

The terminal is what connects the shell to the user. In the olde days of Unix, the terminal would have been a teletype⁵, but nowadays they are usually just a piece of software—a **terminal emulator**—which behaves like one. Modern terminal software often supports displaying different colours, scrolling back through things that were previously displayed,

¹ <mailto:gef@linuxbabble.com>

² http://www.wylug.org.uk/talks/2003/03/unix_command_line.html

³ <http://www.gbdirect.co.uk>

⁴ http://www.wylug.org.uk/talks/2003/03/unix_command_line.pdf

⁵ <http://www.columbia.edu/acis/history/teletype.html>

and displaying a wide variety of characters (example showing Unicode xterm⁶).

There are many terminal emulator programs available. One of the best and most popular is Xterm, which usually comes with X (the Unix graphics subsystem). Xterm is still a good choice, but there are also many alternatives, like the Gnome Terminal (screenshot⁷) and KDE's Konsole (screenshot⁸). These both have more intuitive interfaces for setting preferences. Typically there will be an icon or menu option for launching one of the terminals somewhere on your default desktop.

Special terminal features like colour are enabled through special codes called **escape sequences**. If a program wants to print something in green, it just sends the appropriate codes to the terminal. This is worth knowing, because sometimes the terminal can be left 'messed up', for example if a program crashes. The `reset` command usually gets it back on track.

Your terminal might be connected directly to your shell, or it might be connected over a network with something like `ssh`. The command line is a good interface for administering remote machines. There are ways of running graphical programs over a network, but a text interface is much faster, particularly if the network connection is slow.

The shell

The name 'Bash'

Like many Unix programs, Bash's name is a joke. It stands for 'Bourne again shell', because it's a reincarnation of the original Unix shell by Steve Bourne. That shell was called the Bourne Shell, or `sh` for short.

The term **shell** refers to the 'outer layer' of your software, the part you interact with. In its most general sense it can refer to any kind of user interface software, but under Unix and Linux it always means a command line interface.

By far the most popular shell on Linux is **Bash**. It aims to be compatible with the original Unix shell, `sh`, but adds lots of useful features. There are other shells available, and most Linux distributions will come with at least one other than Bash. The shell is just a normal program which happens to have a special rôle, so it can easily be replaced.

The shell reads the commands you type from the terminal. When you press `Enter`, it interprets a command and performs whatever action is needed to complete it. Usually that means running one or more programs.

Some examples of shell commands

Enough talking, lets see some examples! All the examples in this article use the `$` symbol to represent the prompt your shell prints to the terminal when its waiting for a command, and isn't actually part of the command you type. The actual prompt varies (you can customise it), but it's conventional to use a dollar sign for it in examples.

The `whoami` command is nice and simple—it just tells you your username:

```
$ whoami
geoffr
```

The `cp` command copies a file:

```
$ cp report.txt report-backup.txt
```

To view a PostScript file we can run a program called `gv`. It has a GUI interface, so it will open a separate window to display the document:

```
$ gv unix_command_line.ps
```

The `echo` command simply prints out a message. It doesn't sound very interesting, but it can come in handy:

```
$ echo hello, world
hello, world
```

⁶ `uxterm_utf8_screenshot.png`

⁷ `gnome_terminal_screenshot.png`

⁸ `konsole_screenshot.png`

Filename completion

One of the most common things typed into a shell is the names of files. Commands are available to copy, rename, and delete files, and many other commands process data from files, so its important that filenames are easy to type.

Bash has a feature called **filename completion** to make this easier. If you type the start of a filename and then press Tab, Bash will try to enter the rest of it for you. So If you have a file called *report.txt* you can type a command to view it like this:

```
$ less rep<Tab>
```

Of course, if there are several files whose names start with the letters *rep*, Bash won't know which to choose. It will fill in as much of the filename as possible and then stop (possibly beeping at you). If this happens, press Tab again and Bash will print a list of all the files it thinks you might be referring to. To finish the filename you'll have to type enough of it by hand for the shell to know unambiguously which one you want.

Other completion goodies

Bash's Tab completion also works for commands. If you hit Tab while you're typing the first word on a command line, Bash knows that it's likely to be the name of a program rather than a file in the current directory [?]⁹, so it will try to complete it as such. This can be a good way of stumbling across new commands. Try typing a single letter on the command line and typing Tab twice to get a list of possible completions. If Bash finds many programs which start with that letter it may ask you to confirm that you want to show them—just press *y* to see what programs are available which start with that letter.

Filename completion is also a convenient way of typing filenames which contain 'special' characters. If a filename contains spaces or certain punctuation characters, then they have to be **escaped** to stop Bash from interpreting them specially. The filename completion does this automatically, so you can get away without learning the (fairly complex) rules about how to escape special characters.

If you use Tab on a name which starts with a \$ symbol, Bash will complete shell variable names¹⁰ instead of filenames.

History

The shell has another feature which can help you reduce typing. It keeps a record of all the commands you type, and allows you to go back and rerun them. This is called the **shell history**.

Stored history

Each shell keeps its own history of commands. When the shell exits, it stores these in the file *.bash_history* in your home directory, so the history survives to the next time you run a shell.

To use the history, hit the Up key. The shell should display the last command you entered. You can use Up and Down to go through all the commands you've typed into the current shell. Just press Enter to run the command displayed. Press Page Down to go back down to the 'bottom' of the history if you change your mind.

Some people prefer to use Ctrl+P (for 'previous', the same as Up) and Ctrl+N ('next') to cycle through commands. Control keys can be easier to type, because they allow you to keep your hands over the main part of the keyboard rather than moving to the cursor keys.

The history is particularly useful if you make a mistake in a command. You can go back to it and edit the command. Use the normal editing keys to change the old command and then rerun it.

If you typed a command a long time ago, it can be hard to go back and find it. Bash lets you search the history to make this easier. Type Ctrl+R (for 'reverse search') and then type something you know appears in that command. The commands being found will be displayed as you type. If you don't get the command you wanted, either type some more characters to narrow down the search, or press Ctrl+R again to look for older commands which also match. When you've found the command you want, press Enter to run it, or Left or Right to start editing it.

Globbering

If you want to give lots of filenames to a command, the most convenient way is to have the shell automatically collect them. It can build a list of filenames that match a pattern you give and insert them onto the command line, so that you don't have to type them in by hand.

For example, to delete all the files in the current directory, use the * symbol to collect their names, and the *rm* command to remove them:

⁹ /jargon/current_directory/

¹⁰ /tutorial/command/shell_variable/

```
$ rm *
```

(Be careful: `rm` is a dangerous tool.)

The `*` symbol means “anything is allowed here”. Note that MS-DOS would use `*.*` to match all the files, whereas a Unix shell uses just a single `*`, because on Unix systems filenames don't necessarily contain a `.` character. It is perfectly acceptable for filenames not to have an ‘extension’.

The `*` symbol can be used as part of a more specific pattern to match only some filenames. So to count the number of lines, words, and characters in all the `.txt` files in the current directory (using the `wc` or ‘word count’ command):

```
$ wc *.txt
```

For some reason a pattern like this is known as a **glob**, and so when the shell expands the pattern into a list of filenames it is said to be **globbing**. Unix has more than its fair share of amusing technical terms.

Glob patterns can contain a few other special characters. If a `?` symbol is used, it means “exactly one character is allowed here”. Square brackets work in the same way but only allow specific characters, so `[a-z]` means “any lower-case letter is allowed here”. These are much rarer in normal use than `*`, although they do occasionally come in handy.

Redirection

Many commands allow filenames to be given as options (which is why globbing is useful), and some have options for specifying a file in which to put output. But the shell has a more general mechanism for supplying input and output files, called **redirection**.

The shell can be told to connect a command's input or output to particular files. This is done with the `<` and `>` symbols. So to generate a simple textual calendar for the year 2003 (using the `cal` command) and write it to the file `2003.txt` we can do this:

```
$ cal 2003 >2003.txt
```

Normally the output of the command would be dumped to the terminal, but `>` redirects it into the named file:

The `<` symbol works in the same way as `>`, but causes input to be read from a file. In both cases the symbols point in the direction that the data is flowing. In practice `>` is used far more often than `<`.

Piping

The concept of **piping** is very similar to that of redirection, but instead of connecting one program to a file, piping connects two programs together. Everything that the first program outputs is fed straight into the second program.

A simple (but fairly useless) example of piping is to connect the `echo` command to the `rev` command. We can use `echo` to print out a message (it simply outputs the words on its command line), and feed it into `rev`, which reverses its input:

```
$ echo Happy Birthday | rev
yadhtriB yppaH
```

As for redirection, the shell takes care of setting up the pipe, so the programs don't have to know that anything unusual is happening to their output. When the shell starts the commands running, their relationship looks like this:

Shell scripting

Shells like Bash are actually complete programming languages. It may be that most of the commands you type into a shell are simple commands for manipulating files, but concepts like piping allow them to be used for much more complicated operations. Shell scripting is a way of using the power of the shell to automate common jobs.

A shell script is just a set of shell commands, exactly as they would be typed into a shell. It is basically equivalent to a DOS batch file, but the shells flexibility makes it much more useful. So if you find yourself constructing a complicated pipeline of commands it might be worth copying it into a file so that it can be run again.

Shell scripting is a useful way of building very simple programs, but the shell language is designed to make typing commands easy, and doesn't have the features necessary for complex tasks. When a shell script gets to be more than a few dozen lines long, it may be time to learn Perl...

So...

Unix and Linux can be very powerful from the command line. Modern shells make it possible to get work done effi-

ciently from the command line, without as much typing as you might expect.

Unfortunately, these advantages come at a price. Shells have a steep learning curve, and there are some jobs which will never be easy to do in a purely text-based environment. But for manipulating files and processing text, the shell is often the best choice.

Related Links

- [Linux Babble tutorial on using shell variables and environment variables](#)¹¹

¹¹ [/tutorial/command/shell_variable/](#)